Introduction to Intel-TBB

Cecina Babich Morrow

2024-04-23

The following portfolio contains materials covered in Dr. Christopher Wood's course Parallel Programming with C++.

Introduction to Intel-TBB

Intel-TBB, which stands for Intel's Threading Building Blocks, is a C++ template library for parallel programming. It is now a part of something called oneAPI, which is a set of tools and libraries that Intel has developed to help leverage the power of multi-core processors.

OpenMP vs. Intel-TBB

Both OpenMP and Intel-TBB are third-party tools for parallelization that are not part of C++ itself. OpenMP is a set of compiler directives that you can use to tell the compiler to parallelize your code, while Intel-TBB is a C++ template library that you can use to write parallel code. Intel-TBB uses standard C++ templates and classes, so it is more portable than OpenMP. It is also more efficient. However, it has a steeper learning curve than OpenMP.

Compiling with Intel-TBB

The following code compiles a C++ program using Intel-TBB.

```
g++ -O3 --std=c++14 test.cpp -Iinclude -ltbb -o test
```

Components of the command:

- `-O3` tells the compiler to optimize the code
- `--std=c++14` tells the compiler to use the C++14 standard (you can use other more recent versions if you want)
- `-Iinclude` tells the compiler to look for header files in the `include` directory
- `-ltbb` tells the compiler to link the Intel-TBB library

Functional programming

Intel-TBB is based on functional programming, which is a programming paradigm that treats functions as objects. Functional programming works easily in R and Python since you can use functions as arguments of other functions natively. Things are a bit more complicated with C++ since C++ relies on variable types.

The type of a variable tells you want you can do with that variable and what happens when you act on it. The code referring to a function is a type in C++ just like `int` or `double`, but the syntax gets complicated. For example, the type of a function that takes two `int` arguments and returns an `int` is `int (*)(int, int)`. Fortunately, we can use `auto` instead to avoid needing to write out the type of a function in that way.

Functions as arguments

A template in C++ is a way to write a function or a class without specifying the type of the arguments or the class members. This is useful when you want to write a function or a class that can work with different types of arguments or class members. The following is an example of a template function that takes a function and two arguments and returns the result of calling the function with the two arguments:

```
template<class FUNC, class ARG1, class ARG2>
auto call_function(FUNC func, ARG1 arg1, ARG2 arg2)
{
    auto result = func(arg1, arg2);
    return result;
}
```

In this example, `FUNC` is the type of the function, and `ARG1` and `ARG2` are the types of the arguments. The `auto` keyword tells the compiler to figure out the type of `result` based on the type of `func` and the types of `arg1` and `arg2`.

This means that you can pass function names to other functions. By changing the arguments to a function, you can completely change the function's behavior.

We can use a template to avoid having to write a new function for printing vectors for every type of vector. The following is a function that works with any type of vector:

```
template<class T>
void print_vector(const std::vector<T> &values)
  // const: function doesn't change the vector
  // & is reference: function doesn't make a copy of the vector, but uses the original
{
  std::cout << "[";

  for (const T &value : values)
  {
    std::cout << " " << value;
  }

  std::cout << " ]" << std::endl;
}
```

Mapping / Reducing

Functional programming is based on two main concepts: mapping and reducing. These can be distinguished by the number of inputs and outputs they have:

Output / Input	1	N
1	function	reduce
N	generator	map

Mapping functions

Mapping is the process of applying the same function to every element of a vector (going from N inputs to N outputs). We want to move away from loops, which specify a given order for processing the code. By switching to mapping instead, we are allowing flexibility in the order, which will enable parallel processing.

The following code maps a function to two vectors, showing examples of mapping addition, substraction, and multiplication:

```cpp
#include <iostream>
#include <vector>


template<class FUNC, class T>
auto map(FUNC func, const std::vector<T> &arg1, const std::vector<T> &arg2)
{
  int nvalues = std::min( arg1.size(), arg2.size() );

  auto result = std::vector<T>(nvalues);

  for (int i=0; i<nvalues; ++i)
  {
    result[i] = func(arg1[i], arg2[i]);
  }

  return result;
}

int sum(int x, int y)
{
  return x + y;
}

int difference(int x, int y)
{
  return x - y;
}

int multiply(int x, int y)
{
  return x * y;
}

template<class T>
void print_vector(const std::vector<T> &values)
  // const: function doesn't change the vector
  // & is reference: function doesn't make a copy of the vector, but uses the original
{
  std::cout << "[";

  for (const T &value : values)
  {
    std::cout << " " << value;
  }
  std::cout << " ]" << std::endl;
}

int main()
{
  auto a = std::vector<int>( { 1, 2, 3, 4, 5 } );
  auto b = std::vector<int>( { 6, 7, 8, 9, 10 } );

  auto result = map( sum, a, b );
  print_vector(result);

  result = map( difference, a, b );
  print_vector(result);

  result = map( multiply, a, b );
  print_vector(result);

  return 0;
}
```

We can compile and see the output:

```
g++ --std=c++14 ../intro_to_intel_tbb/workshop/map.cpp -o ../intro_to_intel_tbb/workshop/map
../intro_to_intel_tbb/workshop/map
```

```
## [ 7 9 11 13 15 ]
## [ -5 -5 -5 -5 -5 ]
## [ 6 14 24 36 50 ]
```

The following code maps the `count_lines` function over every file in a folder of Shakespeare plays (see this folder for the files `part1.h` and `filecounter.h`):

```
#include "part1.h"
#include "filecounter.h"

using namespace part1;
using namespace filecounter;

int main(int argc, char **argv) // gives you access to command line arguments
{
  auto filenames = get_arguments(argc, argv);
  auto results = map(count_lines, filenames);

  for (int i = 0; i < filenames.size(); i++)
  {
    std::cout << filenames[i] << " has " << results[i] << " lines." << std::endl;
  }

  return 0;

}
```

We can see the output:

```
../intro_to_intel_tbb/workshop/countlines ../intro_to_intel_tbb/workshop/shakespeare/*
```

```
## ../intro_to_intel_tbb/workshop/shakespeare/allswellthatendswell has 4515 lines.
## ../intro_to_intel_tbb/workshop/shakespeare/antonyandcleopatra has 5998 lines.
## ../intro_to_intel_tbb/workshop/shakespeare/asyoulikeit has 4122 lines.
## ../intro_to_intel_tbb/workshop/shakespeare/comedyoferrors has 2937 lines.
## ../intro_to_intel_tbb/workshop/shakespeare/coriolanus has 5836 lines.
## ../intro_to_intel_tbb/workshop/shakespeare/cymbeline has 5485 lines.
## ../intro_to_intel_tbb/workshop/shakespeare/hamlet has 6045 lines.
## ../intro_to_intel_tbb/workshop/shakespeare/juliuscaesar has 4107 lines.
## ../intro_to_intel_tbb/workshop/shakespeare/kinglear has 5525 lines.
## ../intro_to_intel_tbb/workshop/shakespeare/loveslabourslost has 4335 lines.
## ../intro_to_intel_tbb/workshop/shakespeare/macbeth has 3876 lines.
## ../intro_to_intel_tbb/workshop/shakespeare/measureforemeasure has 4337 lines.
## ../intro_to_intel_tbb/workshop/shakespeare/merchantofvenice has 3883 lines.
## ../intro_to_intel_tbb/workshop/shakespeare/merrywivesofwindsor has 4448 lines.
## ../intro_to_intel_tbb/workshop/shakespeare/midsummersnightsdream has 3115 lines.
## ../intro_to_intel_tbb/workshop/shakespeare/muchadoaboutnothing has 4063 lines.
## ../intro_to_intel_tbb/workshop/shakespeare/othello has 5424 lines.
## ../intro_to_intel_tbb/workshop/shakespeare/periclesprinceoftyre has 3871 lines.
## ../intro_to_intel_tbb/workshop/shakespeare/README has 2 lines.
## ../intro_to_intel_tbb/workshop/shakespeare/romeoandjuliet has 4766 lines.
## ../intro_to_intel_tbb/workshop/shakespeare/tamingoftheshrew has 4148 lines.
## ../intro_to_intel_tbb/workshop/shakespeare/tempest has 3399 lines.
## ../intro_to_intel_tbb/workshop/shakespeare/timonofathens has 3973 lines.
## ../intro_to_intel_tbb/workshop/shakespeare/titusandronicus has 3767 lines.
## ../intro_to_intel_tbb/workshop/shakespeare/troilusandcressida has 5443 lines.
## ../intro_to_intel_tbb/workshop/shakespeare/twelfthnight has 4017 lines.
## ../intro_to_intel_tbb/workshop/shakespeare/twogentlemenofverona has 3605 lines.
## ../intro_to_intel_tbb/workshop/shakespeare/winterstale has 4643 lines.
```

Reductions

A reduction takes a vector as an input and returns a single value, e.g. the sum of all the elements (N inputs to 1 output). The following code shows an example of a reduction function:

```cpp
template<class FUNC, class T>
T reduce(FUNC func, const std::vector<T> &values)
{
    if (values.empty())
    {
        return T();
    }
    else
    {
        T result = values[0];

        for (size_t i=1; i<values.size(); ++i)
        {
            result = func(result, values[i]);
        }
        return result;
    }
}
```

We can use this `reduce` function to sum the line counts of all the files in the Shakespeare folder:

```
#include "part1.h"
#include "filecounter.h"

using namespace part1;
using namespace filecounter;

int main(int argc, char **argv) // gives you access to command line arguments
{
  auto filenames = get_arguments(argc, argv);
  auto results = map(count_lines, filenames);
  auto total = reduce(sum, results);

  std::cout << "Total number of lines: " << total << std::endl;

  return 0;

}
```

We see the output:

```
../intro_to_intel_tbb/workshop/countlines_reduce ../intro_to_intel_tbb/workshop/shakespeare/*
```

```
## Total number of lines: 119685
```

Lambda functions

Lambda functions, also known as anonymous functions, are a way to declare a function without needing to name it and assign it to a variable. They are useful when you want to pass a function as an argument to another function, but you aren't going to be reusing that function and it doesn't make sense to declare it separately. The syntax for lambda functions uses `[](){}`:

- Within `[]` you put the capture list, which specifies which external variables are accessible within the lambda function and whether they are copied by reference or by value
- Within `()` you put the arguments of the function, specifying the types as usual
- Within `{}` you put the body of the function

For instance, the following are equivalent:

```
// traditional way to create function named name
auto name(arguments)
{
    expressions;
}

// unnamed function in lambda function
[](arguments){expressions;}
```

There are various options of what you can put in the capture clause. For instance: + `[=]` means that the lambda function makes a copy of all the variables in the scope of the function where the lambda function is declared + `[&]` means that the lambda function uses the variables in the scope of the function where the lambda function is declared by reference. (It is quicker and uses less memory, but you run the risk of accidentally changing the value of the variable.) + `[x]` captures `x` by value + `[&x]` captures `x` by reference + `[x, &y]` captures `x` by value and `y` by reference

Parallel programming using Intel-TBB

Using this functinal programming style enables us to write code in a way such that it can be run in parallel.

`tbb::parallel_for`

The `tbb::parallel_for` function is a map-type function that allows us to parallelize a for loop. It has the syntax `tbb::parallel_for(range, function)`, where `range` is the set of values over which we want to iterate, and `function` is the function to apply to each subset of range values. The following code shows an example of using `tbb::parallel_for`:

```cpp
#include <iostream>
#include <vector>
#include <cmath>
#include <tbb/parallel_for.h>

int main(int argc, char **argv)
{
  auto values = std::vector<double>(10000);

  tbb::parallel_for(

    // Entire range of values
    tbb::blocked_range<int>(0, values.size()),

    // This is what each thread will do:
    [&](tbb::blocked_range<int> r){
      for (int i=r.begin(); i<r.end(); ++i)
      {
        values[i] = std::sin(i * 0.001);
      }
    }

  );

  double total = 0;

  for (double value : values)
  {
    total += value;
  }

  std::cout << total << std::endl;
  return 0;
}
```

`tbb::blocked_range` is a class that represents a range of values which can be divided into smaller ranges that can be processed in parallel. Here, it defines the range of indices from `0` to `values.size()` for processing in parallel. The lambda function (`[&](tbb::blocked_range<int> r) {...}`) is then executed in parallel for a subrange of indices `r`.

We can compile and see the output as follows:

```
g++ --std=c++14 ../intro_to_intel_tbb/workshop/parallel_for.cpp -Iinclude -ltbb -o \
  ../intro_to_intel_tbb/workshop/parallel_for
../intro_to_intel_tbb/workshop/parallel_for
```

```
## 1839.34
```

Note that after computing each element of `values` in parallel, we add them to `total` one at a time using a for loop to avoid multiple threads trying to read / write to `total` concurrently. If we wanted to parallelize this loop as well, we would need to use a parallel form of a reduction.

`tbb::parallel_reduce`

`tbb::parallel_reduce` is the reduction counterpart to `tbb::parallel_for`. It has the syntax `tbb::parallel_for(range, identity_value, function, reduction function)`, where:

- `range` is the same as in `tbb::parallel_for`
- `identity_value` is the starting value for reduction, e.g. typically 0.0 for addition or 1.0 for multiplication
- `function` is a lambda function as in `tbb::parallel_for`
- `reduction_function` is the function being used to reduce the values, e.g. `std::plus<double>()` to reduce doubles using addition

We can now edit our code from earlier to parallelize the loop for calculating `total` as well:

```
#include <iostream>
#include <vector>
#include <cmath>
#include <tbb/parallel_for.h>

int main(int argc, char **argv)
{
  auto values = std::vector<double>(10000);

  tbb::parallel_for(
    tbb::blocked_range<int>(0, values.size()),
    [&](tbb::blocked_range<int> r){
      for (int i=r.begin(); i<r.end(); ++i)
      {
        values[i] = std::sin(i * 0.001);
      }
    }
  );

  auto total = tbb::parallel_reduce(
    tbb::blocked_range<int>(0,values.size()),
    0.0,
    [&](tbb::blocked_range<int> r, double running_total)
    {
      for (int i=r.begin(); i<r.end(); ++i)
      {
        running_total += values[i];
      }

      return running_total;
    }, std::plus<double>() );

  std::cout << total << std::endl;

  return 0;
}
```

Compiling to check our work:

We can compile and see the output as follows:

```
g++ --std=c++14 ../intro_to_intel_tbb/workshop/parallel_reduce.cpp -Iinclude -ltbb -o \
  ../intro_to_intel_tbb/workshop/parallel_reduce
../intro_to_intel_tbb/workshop/parallel_reduce
```

```
## 1839.34
```